

Note: This PDF is a sample chapter from the full book - "Android Internals: A confectioner's cookbook" - which can be found on <http://NewAndroidBook.com/>. The chapter was made available for free as part of an AnDevCon 2014 lecture, which is itself a subset of Technologeeks.com [Android Internals training](#). You can also preorder the book by emailing preorder@NewAndroidBook.com.

Note some links (to other chapters in the book) will not work in this PDF, but external links will. Feedback, questions and requests are always welcome.

XXI: Android Security

As with other operational aspects, Android relies on the facilities of Linux for its basic security needs. For most apps, however, an additional layer of security is enforced by the Dalvik Virtual Machine. Android Security is therefore an amalgam of the two approaches - VM and native - which allows for defense in depth.

This chapter starts by exploring the Linux user model, and its adaptation to the Android landscape. Starting with the native Linux permissions, and the clever usage of IDs for Apps and group membership. We then proceed to highlight capabilities, an oft overlooked feature of Linux used extensively in Android to work around the inherent limitation using the almighty root uid in the classic model. Next is a discussion of [SELinux](#), a Mandatory Access Control (MAC) framework introduced in 4.3 and enforced in 4.4. Lastly, we consider various protections against code injection, the bane of application security.

At the Dalvik level, we consider the simple, yet effective permission model enforced by the Virtual Machine, as well as the little known mechanism of [Intent Firewall](#). Binder was covered in the [previous chapter](#), but the high level view of security doesn't require you to read it as a prerequisite. Up to this point, the discussion can be thought of as aspects of *application level* security.

Next, we consider *user-level* security: protecting the device against human users by [locking the device](#). No longer the domain of simple PINs and patterns, device locking methods get ever more innovative, and have expanded to include biometrics as well. As of JB, Android allows multiple users to coexist, each with his or her own private data, and set of installed applications, and so the implementation of multiple users [is covered as well](#).

At this point, we turn to a discussion of encryption on Android. Beginning with aspects of key management, we explain the inner workings of the keystore service, and the maintenance of certificates on the device. We then touch on Android's [storage encryption feature](#) (introduced in HoneyComb) and filesystem authentication using [Linux's dm-verity](#) (as introduced in KitKat).

Last, but in no way least, is a focus on [device rooting](#), without which no discussion about security would be complete. Rooting brings with it tremendous advantages to the power user (and is one of the reasons Android's popularity has exploded in hacker and modder circles), but also woeful, dire implications on application and system security. The two primary methods - boot-to-root and "one-click" are detailed and contrasted.

Security at the Linux Level

Android builds a rich framework on top of the Linux substrate, but at its core, relies on Linux for virtually all operations. The Linux inheritance also binds Android to use the same security features as those offered by Linux - the permissions, capabilities, SELinux, and other low-level security protections.

Linux Permissions

The security model of Linux is a direct port of the standard UN*X security model. This model, which has remained largely unchanged since its inception some 40 years ago, provides the following primitives:

- **Every user has a numeric user id**: The actual user name doesn't matter, though some usernames are reserved for system users (which are designated the owners of configuration files and directories). Two users may share the same user id, but this in effect means that, as far as the system is concerned, this represents a single user with two username/password combinations.
- **Every user has a numeric primary group id**: Much like the username, the group name doesn't matter, and some GIDs are reserved for system use.
- **Users may hold memberships in additional groups**: Traditionally, additional group memberships is maintained by the `/etc/group` file. It lists the group names, group ids, and any members who are not already in a group by virtue of the primary GID.
- **Permissions on file are granted for a specified user, group, and "other"**: This is the familiar output of `"ls -l"`, which maps the permissions (read, write or execute) to the user and group, and the "rest of the world". Both files and directories follow this extremely limited model, for which UN*X has been duly criticized. Because of its limitations, file access requirements basically force the creation of specialized groups
- **(Almost) everything in UN*X can be accessed as files**: It thus follows that access to system resources - named IPC objects, UNIX domain sockets, and devices - is a corollary of file permissions. In other words, since the resources have a filesystem representation they can be `chown/chgrp/chmod`d just as files can be, and have the same type of permissions.
- **UID 0 is omnipotent**: Because of the way permission checks are implemented, "0" effectively short circuits the checks and grants access to all files, or resource. What follows is that uid 0 (the "root" user) wields power absolute over the system.
- **SetUID or SetGID binaries allow assuming another uid (or joining another group) during their execution**: with no questions asked. Having execute permission to a `Set[ug]id` binary will automatically bestow those special permissions. This mechanism, which rightfully looks like a gaping design flaw, is actually a feature, used to work around privileged operations, such as changing one's uid (`su`) or password (`passwd`). Such operations - by definition - are only possible for uid 0, but can be enabled if the root user empowers specific binaries (by `chmod 4xxx` and `2xxx`, for `SetUID` and `SetGID`, respectively). As a precaution, copying or moving the binaries will strip those bits.

Android takes the classic model - which it obtains for free from the underlying Linux system - and naturally employs it, but offers a different, somewhat novel interpretation: In it, the "users" are granted to individual applications, not human users. Suddenly, much in the same way as human users sharing the same UN*X server were compartmentalized from one another, applications enjoy (and are limited by) the same seclusion. A user cannot access another user's files, directories, or processes - and this exact isolation enables applications to run alongside each other, but with no power to influence one another. This approach is quite unique to Android - iOS runs all applications under one uid (mobile, or 501) and relies on kernel-enforced sandboxing to isolate applications from one another.

When an application is installed for the first time, the `PackageManager` assigns it a unique user id - which is understandably referred to as an *application id*. This id is taken from the range of 10000-90000, and bionic - the Android C runtime library - automatically maps this to a human readable name - `app_XXX` or `u_XXXX`.

Android can't get rid of `SetUID` support entirely - because this requires recompilation of the kernel and other modifications. Beginning with JB 4.3, however, no `SetUID` binaries are installed by default, and the `/data` partition is mounted with the `nosuid` option.

System defined AIDs

Android maintains the lower range of user ids - 1000-9999 - exclusive for system use. Only a subset of this range is actually used, and it is hardcoded in [android_filesystem_config.h](#). Table 21-1 shows the UIDs defined and used by Android. Most of these are used as GIDs as well: By joining secondary groups, system processes like `system_server`, `adb`, `install` and others gain the ability to access system files and devices, which are owned by these groups - a simple yet effective strategy.

Table 21-1: Android AIDs and their default holders

GID	#define	Members	Permits
1001	AID_RADIO	system_server	/dev/socket/rild (To Radio Interface Layer Daemon) Access net.*, radio.* properties
1002	AID_BLUETOOTH	system_server	Bluetooth configuration files
1003	AID_GRAPHICS	system_server	/dev/graphics/fb0, the framebuffer
1004	AID_INPUT	system_server	/dev/input/*, the device nodes for input devices.
1005	AID_AUDIO	system_server	/dev/eac, or other audio device nodes access /data/misc/audio, read /data/audio
1006	AID_CAMERA	system_server	Access to camera sockets
1007	AID_LOG	system_server	/dev/log/*
1008	AID_COMPASS	system_server	Compass and location services
1009	AID_MOUNT	system_server	/dev/socket/vold, on the other side of which is the VOLUME Daemon
1010	AID_WIFI	system_server	WiFi Configuration files (/data/misc/wifi)
1011	AID_ADB	(reserved)	Reserved for ADBD. Owns /dev/android_adb.
1012	AID_INSTALL	installd	Owns some application data directories
1013	AID_MEDIA	mediaserver	Access /data/misc/media, and media.* service access
1014	AID_DHCP	dhcpcd	Access /data/misc/dhcp Access dhcp properties
1015	AID_SDCARD_RW		Group owner of emulated SDCard
1016	AID_VPN	mtpd racoon	/data/misc/vpn, /dev/ppp
1017	AID_KEYSTORE	keystore	Access /data/misc/keystore (system keystore)
1018	AID_USB	system_server	USB Devices
1019	AID_DRM		Access to /data/drm
1020	AID_MDNSR	mdnsd	Multicast DNS and service discovery
1021	AID_GPS		Access /data/misc/location
1023	AID_MEDIA_RW	sdcard	Group owner of /data/media and real SDCard
1024	AID_MTP		MTP USB driver access (not related to mtpd)
1026	AID_DRMRPC		DRM RPC
1027	AID_NFC	com.android.nfc	Near Field Communication support: /data/nfc, and nfc service lookup
1028	AID_SDCARD_R		external storage read access
1029	AID_CLAT		CLAT (IPv6/IPv4)
1030	AID_LOOP_RADIO		Loop Radio devices
1031	AID_MEDIA_DRM		DRM plugins. Access to /data/mediadrn.
1032	AID_PACKAGE_INFO		Package information metadata
1033	AID_SDCARD_PICS		PICS folder of SD Card
1034	AID_SDCARD_AV		Audio/Video folders of SD Card
1035	AID_SDCARD_ALL		All SDCard folders

Android system properties also rely on UIDs for access control - init's property_service limits access to several property namespaces, as was shown in [Chapter 4](#). It likewise falls on the servicemanager, as the crux of all IPC, to provide basic security. Though the Binder eventually provides security through a uid/pid model, servicemanager can restrict the lookup of well known service names to given uids through a hard-coded allowed array, though uid 0 or SYSTEM are always allowed to register. This is shown in Listing 21-1:

Listing 21-1: Hard-coded services permissions (from [service_manager.c](#))

```
/* TODO:
 * These should come from a config file or perhaps be
 * based on some namespace rules of some sort (media
 * uid can register media.*, etc)
 */
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
    { AID_MEDIA, "media.audio_flinger" },
    { AID_MEDIA, "media.log" },
    { AID_MEDIA, "media.player" },
    { AID_MEDIA, "media.camera" },
    { AID_MEDIA, "media.audio_policy" },
    { AID_DRM, "drm.drmManager" },
    { AID_NFC, "nfc" },
    { AID_BLUETOOTH, "bluetooth" },
    { AID_RADIO, "radio.phone" },
    { AID_RADIO, "radio.sms" },
    { AID_RADIO, "radio.phonesubinfo" },
    { AID_RADIO, "radio.simphonebook" },
/* TODO: remove after phone services are updated: */
    { AID_RADIO, "phone" },
    { AID_RADIO, "sip" },
    { AID_RADIO, "isms" },
    { AID_RADIO, "iphonesubinfo" },
    { AID_RADIO, "simphonebook" },
    { AID_MEDIA, "common_time.clock" },
    { AID_MEDIA, "common_time.config" },
    { AID_KEYSTORE, "android.security.keystore" },
};
// .....
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM))
        return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;

    return 0;
}
```

With the introduction of SE-Linux, and the slow but steady migration of Android to it, it's likely that the hard-coded method will be soon superseded in favor of integration with an SE-Linux policy, much in the same way as `init`'s properties have. At any rate, it's important to note this is but one layer of security: `servicemanager` refuses to allow untrusted AIDs to register well known names. As we discuss later, the Binder allows both client and server to perform [additional permission checks](#), and an additional layer of [Dalvik-level permissions](#) is also employed.

Paranoid Android GIDs

Android GIDs of 3000 through 3999 are also recognized by the kernel, when the `CONFIG_PARANOID_ANDROID` is set. This restricts all aspects of networking access to these GIDs only, by enforcing additional gid checks in the kernel socket handling code. Note that `netd` overrides these settings, because it is running as root. Table 21-2 shows the known network ids

Table 21-2: Android Network-related AIDs and their holders

GID	#define	Members	Permits
3001	AID_BT_ADMIN	system_server	Creation of AF_BLUETOOTH sockets
3002	AID_NET_BT	system_server	Creation of sco, rfcomm, or l2cap sockets
3003	AID_NET_INET	system_server	/dev/socket/dnsproxyd, and AF_INET[6] (IPv4, IPv6) sockets
3004	AID_NET_RAW	system_server, mtpd, mdnsd	Create raw (non TCP/UDP or multicast) sockets
3005	AID_NET_ADMIN	racoond, mtpd	Configure interfaces and routing tables
3006	AID_NET_BW_STATS	system_server	Reading bandwidth statistics accounting
3007	AID_NET_BW_ACCT	system_server	Modifying bandwidth statistics accounting

Isolated Services

As of Jelly Bean (4.1) Android introduces the notion of isolated services. This feature is a form of compartmentalization (similar to iOS's XPC) which enables an application to run its services in complete separation - in a different process, with a separate UID. Isolated services use the UID range of 99000 through 99999 (AID_ISOLATED_START through _END), and the servicemanager will deny them any request. As a consequence, they cannot lookup any system services, and are effectively limited to in memory operations. This is primarily useful for applications such as web browsers, and indeed Chrome is a prime example of using this mechanism. As shown in output 21-1, isolated services are marked as u##_i##:

Output 21-1: Chrome's isolated services

```
shell@htc_m8wl:/ $ ps | grep chrome
u0_a114  4577  384  1178728 118528 ffffffff 4007941c S com.android.chrome
u0_i0    5510  384  1283624 89788  ffffffff 4007941c S com.android.chrome:sandboxed_process0
#
# Pulling the Chrome.apk to the host and dumping its manifest:
#
morpheus@Forge (/tmp)$ /aapt d xmltree Chrome.apk AndroidManifest.xml
....
E: service (line=285)
  A: android:name(0x01010003)="org.chromium.content.app.SandboxedProcessService0"
  A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
  A: android:exported(0x01010010)=(type 0x12)0x0
  A: android:process(0x01010011)=":sandboxed_process0"
  A: android:isolatedProcess(0x010103a9)=(type 0x12)0xffffffff 0xffffffff="true", so isolated
# ... 12 more entries
E: service (line=298)
  A: android:name(0x01010003)="org.chromium.content.app.PrivilegedProcessService0"
  A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
  A: android:exported(0x01010010)=(type 0x12)0x0
  A: android:process(0x01010011)=":privileged_process0"
  A: android:isolatedProcess(0x010103a9)=(type 0x12)0x0 0x0="false", so not isolated
...
```

Root-owned processes

As with Linux, the root user - uid 0 - is still just as omnipotent - but far from omnipresent: Its use is limited to the absolute bare minimum, and that minimum is shrinking from one Android release to another. Quite a few previous Android exploits targetted root-owned processes (with vold being a perennial favorite), and the hope is that by reducing their number, the attack surface could be greatly reduced. The `install.d` is an example of such a process, whose root privileges have been removed beginning with JellyBean.

It is likely impossible to remove all root owned processes: At the very least, `init` needs to retain root capabilities, as does `Zygote` (whose `fork()` assume different uids, something only uid 0 can do). You can see the root owned processes on your device by typing `ps | grep ^root | grep -v " 2 "` (The `grep -v` ignores kernel threads, whose PPID is 2). Table 21-3 shows the services which still run as root by default in KitKat (but note your device may have more, as added by the device vendor)

Table 21-3: Android services still running as root

Service	Rationale
init	Somebody <i>has</i> to maintain root privileges in the system and launch others - might as well be PID 1
healthd (init)	Minimal operation
ueventd (init)	Minimal operation
zygote	Requires <code>setuid()</code> to change into AID when loading APKs, retains capabilities for <code>system_server</code>
adb	Developers may need legitimate root access; system trusts ADB to immediately drop privileges to <code>shell</code> if <code>ro.debuggable</code> is 0 or <code>ro.secure</code> is 1
vold	[Un/]Mounting filesystems, and more.
netd	Configuring interfaces, assigning IPs, DHCP and more
mpdecision	Multi core scheduler control (technically, not part of AOSP, but nonetheless common)

Eventually, it is expected that Android will leave only those services which absolutely **must** have root, and others will follow in the steps of `install.d`. To do so, Android will have to increase its usage of another important Linux security feature - Capabilities.

Linux Capabilities

Originally part of the POSIX.1e draft (and thus meant to be incorporated as a standard for all UN*X), capabilities were an early adoption into the 2.2 line of kernels. Though the POSIX draft was eventually withdrawn, capabilities remained implemented in Linux, and have since been expanded and improved on. Distributions of Linux don't make use of capabilities all that often, but Android makes extensive use of them.

The idea behind capabilities is to break the "all-or-nothing" model of the root user: The root user is fully omnipotent, whereas all other users are, effectively, impotent. Because of this, if a user needs to perform some privileged operation, the only standard solution is to resort to SetUID - become uid 0, for the scope of the operation, then yield superuser privileges, and revert to a non-privileged user.

If a SetUID binary can be trusted, then - in theory - the model should work. In practice, however, SetUID poses inherent security risks: If a SetUID binary is somehow exploited, it could be tricked into compromising root. Common tricks include symlinks and race conditions (diverting the binary to overwrite system configuration files), and code injection (forcing the binary to execute a root shell - hence the term "shellcode" for injected code).

Capabilities offer a solution to this problem, by "slicing up" the powers of root into distinct areas, each represented by a bit in a bitmask, and allowing or restricting privileged operations in these areas only, by toggling the bitmask. This makes them an implementation of the *principle of least privilege*, a tenet of security which dictates that an application or user must not be given any more rights than are absolutely required for its normal operation. This way, even if the application or user ends up being malicious, its scope of damage is compartmentalized to only those operations which it could have done anyway - but it cannot run amuck and compromise the system security. In fact, a nice side effect of capabilities is that they can be used to restrict the root user itself, in cases where the user behind the uid is not fully trustworthy.

init still starts most of Android's server processes as root, and these processes have the full capabilities bitmask (0xffffffffffffffff) as they launch. Before these processes actually do anything, however, they drop their privileges, and retain only the capabilities they need. A good example can be seen in installd:

Listing 21-2: Installd's usage of capabilities

```
static void drop_privileges() {

    // Ask the kernel to retain capabilities, since we setgid/setuid next
    if (prctl(PR_SET_KEEPCAPS, 1) < 0) {
        ALOGE("prctl(PR_SET_KEEPCAPS) failed: %s\n", strerror(errno));
        exit(1);
    }

    // Switch to gid 1012
    if (setgid(AID_INSTALL) < 0) {
        ALOGE("setgid() can't drop privileges; exiting.\n");
        exit(1);
    }

    // Switch to uid 1012
    if (setuid(AID_INSTALL) < 0) {
        ALOGE("setuid() can't drop privileges; exiting.\n");
        exit(1);
    }

    struct __user_cap_header_struct capheader;
    struct __user_cap_data_struct capdata[2];
    memset(&capheader, 0, sizeof(capheader));
    memset(&capdata, 0, sizeof(capdata));
    capheader.version = _LINUX_CAPABILITY_VERSION_3;
    capheader.pid = 0;

    // Request CAP_DAC_OVERRIDE to bypass directory permissions
    // Request CAP_CHOWN to change ownership of files and directories
    // Request CAP_SET[UG]ID to change identity
    capdata[CAP_TO_INDEX(CAP_DAC_OVERRIDE)].permitted |= CAP_TO_MASK(CAP_DAC_OVERRIDE);
    capdata[CAP_TO_INDEX(CAP_CHOWN)].permitted        |= CAP_TO_MASK(CAP_CHOWN);
    capdata[CAP_TO_INDEX(CAP_SETUID)].permitted        |= CAP_TO_MASK(CAP_SETUID);
    capdata[CAP_TO_INDEX(CAP_SETGID)].permitted        |= CAP_TO_MASK(CAP_SETGID);

    capdata[0].effective = capdata[0].permitted;
    capdata[1].effective = capdata[1].permitted;
    capdata[0].inheritable = 0;
    capdata[1].inheritable = 0;


    if (capset(&capheader, &capdata[0]) < 0) { ALOGE("capset failed: %s\n", strerror(errno));
}
}
```

The heaviest user of capabilities is, unsurprisingly, `system_server`, since it is a system owned process, but still needs root privileges for many of its normal operations. Table 21-4 shows the Linux capabilities, and the Android processes known to use them:

Table 21-4: Linux capabilities used by Android processes

capability	#define	Users	Permits
0x01	CAP_CHOWN	installd	Change file and group ownership
0x02	CAP_DAC_OVERRIDE	installd	Override Discretionary Access Control on files/dirs
0x20	CAP_KILL	system_server	Kill processes not belonging to the same uid
0x40	CAP_SETGID	installd	allow setuid(2), seteuid(2) and setfsuid(2)
0x80	CAP_SETUID	installd	allow setgid(2) and setgroups(2)
0x400	CAP_NET_BIND_SERVICE	system_server	Bind local ports at under 1024
0x800	CAP_NET_BROADCAST	system_server	Broadcasting/Multicasting
0x1000	CAP_NET_ADMIN	system_server	Interface configuration, Routing Tables, etc.
0x2000	CAP_NET_RAW	system_server	Raw sockets
0x10000	CAP_SYS_MODULE	system_server	Insert/remove module into kernel
0x800000	CAP_SYS_NICE	system_server	Set process priority and affinity
0x1000000	CAP_SYS_RESOURCE	system_server	Set resource limits for processes
0x2000000	CAP_SYS_TIME	system_server	Set real-time clock
0x4000000	CAP_SYS_TTY_CONFIG	system_server	Configure/Hangup tty devices
0x400000000	CAP_SYSLOG	dumpstate	Configure kernel ring buffer log (dmesg)

Note, that table 21-4 provides a limited (albeit large) subset of the Linux capabilities. It is likely that over the evolution of both Linux and Android more capabilities will be added. The following experiment demonstrates how you can see capabilities used by processes:



Experiment: Viewing capabilities and group memberships

You can easily view `system_server`'s capabilities and group memberships (or those of any other process, for that matter), by looking at `/proc/${PID}/status`, replacing `${PID}` with the pid of the process in question:

Output 21-2: Viewing `system_server`'s capabilities and group memberships

```

root@generic:/ # cat /proc/${SS_PID}/status
Name:    system_server
State:   S (sleeping)
Tgid:    372
Pid:     372
PPid:    52
TracerPid: 0
Uid:     1000    1000    1000    1000 # AID_SYSTEM
Gid:     1000    1000    1000    1000 # AID_SYSTEM
...      # Note the secondary group memberships, below
Groups:  1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1032 3001 3002 3003 3006 3007
...
CapInh:  0000000000000000 # Inherited
CapPrm:  0000000007813c20 # Permitted
CapEff:  0000000007813c20 # Effective
CapBnd:  ffffffff00000000 # Bounding set
...

```

In the above, you can see four bitmasks for capabilities: Those inheritable by child process, those potentially permitted for this process, those actively in effect (as in, permitted and also explicitly required by the process), and the bounding set. The bounding set (added in Linux 2.6.25) is a bitmask which limits the usage of `capset(2)`.



Experiment: Viewing capabilities and group memberships (cont.)

By looking over PIDs in `/proc`, you can single out the processes which use capabilities. This requires a bit of shell scripting, as shown in the following output:

Output 21-3: Processes with capabilities

```
root@htc_m8wl:/proc # for p in [0-9]*;          \ # Iterate over all PIDs
do CAP=`grep CapPrm $p/status | \ # Get permitted capabilities
grep -v -v ffffff | \ # Rule out root processes
grep -v 0000000000000000`; \ # Rule out incapable processes
if [[ ! -z $CAP ]]; then \ # If capabilities were found,
grep Name $p/status; \ # Print the process name
echo PID $p - $CAP; \ # PID, and capabilities mask
fi;
done
Name: system_server
PID 13662 - CapPrm: 00000000007813c20
Name: wpa_supplicant
PID 13907 - CapPrm: 0000000000003000
Name: rild
PID 368 - CapPrm: 0000000000003000
Name: netmgrd
PID 375 - CapPrm: 0000000000003000
Name: installld
PID 387 - CapPrm: 00000000000000c3
Name: dumpstate
PID 389 - CapPrm: 0000000400000000
Name: dumpstate
PID 390 - CapPrm: 0000000400000000
Name: qseecomd
PID 398 - CapPrm: 0000000000222000
Name: qseecomd # A secondary thread of qseecomd, therefore same capabilities
PID 510 - CapPrm: 0000000000222000
```

As the above show, the capabilities are in line with Table s2-ssc. Note that some vendors (above, HTC) may add their own processes (above, `qseecomd`) with additional capabilities.

Beginning with JB (4.3), Zygote calls `prctl(PR_CAPBSET_DROP)` and `prctl(PR_SET_NO_NEW_PRIVS)`, to ensure that no further capabilities can be added to its child processes (i.e. the user apps). It is likely that, going forward, `vold` and `netd` will both drop their privileges and rely on capabilities, rather than retain their root privileges. This is especially important considering `vold`'s history of vulnerabilities.

SELinux

SELinux - Security Enhanced - marks a step further in the evolution of Linux beyond standard UNIX. Originally developed by the NSA, SELinux is a set of patches which have long since been incorporated into the mainline kernel, with the aim of providing a Mandatory Access Control (MAC) framework, which can restrict operations to a predefined policy. As with capabilities, SELinux implements the principle of least privilege, but with much finer granularity. This greatly augments the security posture of a system, by preventing processes from operating outside strictly defined operational bounds. So long as the process is well behaving, this should pose no problem. If the process misbehaves, however (as most often is the case of malware, or the result of code injection), SELinux will block any operation which exceeds those bounds. The approach is very similar to iOS's sandbox (which builds on the TrustedBSD MAC Framework), though the implementation is quite different.



Though long included in Linux (and, like capabilities, not always implemented by default), SELinux was introduced into Android with JellyBean (4.3). The initial introduction was gentle - setting SELinux in permissive mode, wherein any violations of the policy are merely audited. With KitKat (4.4), however, SELinux now defaults to enforcing mode for several of Android's services (specifically, `installd`, `netd`, `vold` and `zygote`), though still permissive for all other processes. In general, it is considered a good practice to use the per-domain permissive mode, in order to test a policy before setting it to enforcing, and it is likely that enforcement will expand with the next version of Android.

SELinux's port to Android - commonly referred to as SEAndroid - was first described in [a paper](#) and [a presentation](#) by Smalley and Craig of the NSA (who have followed up on SEAndroid with an excellent presentation in [the 2014 Android Builders Summit](#)). Google provides basic documentation in the [Android Source site](#). Of the mainline Linux distributions RedHat has been an early adopter, and provides a [comprehensive guide](#).

SEAndroid follows the same principle of the original, but extends it to accommodate Android specific features - such as system properties, and (naturally) the Binder (via kernel hooks). Samsung further extends SEAndroid, and uses it as a foundation for their "KNOX" secure platform (currently in v2.0). KNOX boasts a stronger security policy, enforcing and confining all processes (except `init` and the kernel threads). In the following discussion, "SELinux" refers to those features found in both Linux and Android, whereas "SEAndroid" refers only to the latter.

The main principle of SELinux (and, in fact, most MAC frameworks) is that of *labeling*. A label assigns a type to a resource (object), and a security **domain** for a process (subject). SELinux can then enforce so as to allow only processes in the same domain (likewise labeled) to access the resource (Some MAC Frameworks go as far as to make resources with different labels invisible, somewhat akin to the Linux concept of namespaces, although SELinux does not go that far). Depending on the policy, domains can also be made **confined**, so that processes cannot access any resource but those allowed. The policy enforcement is performed independently of other layers of permissions (e.g. file ACLs). The policy may also allow relabeling for some labels (`relabelto` and `relabelfrom`, also called a **domain transition**) in some cases, which is a necessary requirement if a trusted process (e.g. `Zygote`) spawns an untrusted one (virtually any user application).

An SELinux label is merely a 4-tuple, formatted as a string of the form `user:role:type:level`. All processes with the same label (i.e. in the same domain) are equivalent. SEAndroid (presently) only defines the *type* - i.e. the label is always in the form `u:r:domain:s0`. As of KitKat, the SEAndroid policy defines individual domains for all daemons (i.e. each daemon gets its own permissions and security profiles), along with the domains shown in table 21-5, for application classes.

Table 21-5: The application class domains in Android 4.4

Label (domain)	Apps	Restrictions
<code>u:r:kernel:s0</code>	Reserved for kernel threads	Unconfined
<code>u:r:isolated_app:s0</code>	Isolated processes	Previously connected anonymous UNIX sockets, read/write
<code>u:r:media_app:s0</code>	signed with media key	Allowed to access network
<code>u:r:platform_app:s0</code>	signed with platform key	
<code>u:r:shared_app:s0</code>	signed with shared key	
<code>u:r:release_app:s0</code>	signed with release key	
<code>u:r:untrusted_app:s0</code>	All other	Access ASEC, SDCard, TCP/UDP sockets, PTYs

The keys referred to in table 21-5 are defined in `/system/etc/security/mac_permissions.xml`, which is part of the **middleware MAC** (MMAC) implementation: The Package Manager recognizes the keys used for signing apps, and labels the applications accordingly (using a call to `SELinuxMMAC.assignSeinfoValue`). This is done during package scanning (part of the package installation, as described in [Chapter 9](#)). Note the term *middleware* here applies to labeling performed strictly in user mode by the Android system components.

All the `_app` domains inherit from the base `appdomain`, which allows the basic application profile, including actions such as using the binder, communicating with `zygote`, `sufaceflinger`, etc. You can find the type enforcement (`.te`) files, which contain the detailed definitions for all domains, in the AOSP's `external/sepolicy` directory. The syntax used in those files is a mixture of keywords and macros (from `temacros`), which allow or deny operations in the domain, as shown in Listing 21-3:

Listing 21-3: Sample `te` file (`debuggerd.te`)

```
# debugger interface
type debuggerd, domain;
permissive debuggerd;
type debuggerd_exec, exec_type, file_type;

init_daemon_domain(debuggerd)
unconfined_domain(debuggerd)
relabelto_domain(debuggerd)
allow debuggerd tombstone_data_file:dir relabelto;
```

force automatic transition when init spawns us
Leaves debuggerd unconfined, at present
Allow domain transition to this domain
Allow debugged to manipulate tombstone files

The files in `external/sepolicy` form the baseline, which all devices are meant to automatically inherit from. Rather than modify them, vendors are encouraged to add four* specific variables in their `BoardConfig.mk` file, specifying `BOARD_SEPOLICY_[REPLACE|UNION|IGNORE]`, to override, add or omit files from the policy, and `BOARD_SEPOLICY_DIRS` to provide the search path for the directories containing their files. This mitigates the risk of an accidental policy change due to file error, which may result in security holes. The directory also contains the `mac_permissions.xml` template, which is populated with keys in `keys.conf` (shown in Listing 21-4):

Listing 21-4: The `mac_permissions.xml` and `keys.conf` files

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
  <!-- Platform dev key in AOSP -->
  <signer signature="@PLATFORM" >
    <seinfo value="platform" />
  </signer>

  <!-- Media dev key in AOSP -->
  <signer signature="@MEDIA" >
    <seinfo value="media" />
  </signer>

  <!-- shared dev key in AOSP -->
  <signer signature="@SHARED" >
    <seinfo value="shared" />
  </signer>

  <!-- release dev key in AOSP -->
  <signer signature="@RELEASE" >
    <seinfo value="release" />
    <!-- %lt; /signer -->
  </signer>

  <!-- All other keys -->
  <default>
    <seinfo value="default" />
  </default>
</policy>
```

```
[ @PLATFORM ]
ALL : platform.x509.pem

[ @MEDIA ]
ALL : media.x509.pem

[ @SHARED ]
ALL : shared.x509.pem

# Example of ALL TARGET_BUILD_VARIANTS
[ @RELEASE ]
ENG      : testkey.x509.pem
USER     : testkey.x509.pem
USERDEBUG : testkey.x509.pem
```

The stock type enforcement files are all concatenated and compiled into the resulting `/sepolicy` file, which is a binary file placed on the root file system. Doing so offers further security, because the root filesystem is mounted from the `initramfs`, which is itself part of the `bootimg`, that is digitally signed (and therefore hopefully tamperproof). The compilation is performed merely as an optimization, and the resulting file can be easily decompiled, as is shown in the experiment `sec-dispol`. The binary policy file can be loaded through `/sys/fs/selinux` (though `init` most commonly does so through `libselinux.so`).



Experiment: Decompiling an Android /sepolicy file

If you have a Linux host, decompiling an /sepolicy can be performed with the `sedispol` command, which is part of the `checkpolicy` package. Assuming Fedora or a similar derivative, this first involves getting the package, if you don't already have it:

Output 21-4: Obtaining the checkpolicy package

```
root@Forge (~)# yum install checkpolicy
Loaded plugins: langpacks, refresh-packagekit
--> Running transaction check
--> Package checkpolicy.x86_64 0:2.1.12-3.fc19 will be installed
--> Finished Dependency Resolution
..
Total download size: 245 k
Installed size: 1.0 M
Is this ok [y/d/N]: y
..
Installed:
  checkpolicy.x86_64 0:2.1.12-3.fc19
root@Forge (~)# rpm -ql checkpolicy
/usr/bin/checkmodule
/usr/bin/checkpolicy
/usr/bin/sedismod
/usr/bin/sedispol # This is the policy disassembler
/usr/share/man/man8/checkmodule.8.gz
/usr/share/man/man8/checkpolicy.8.gz
```

Once you have the command, all you need is to transfer the policy to the host, and start examining it (the command is an interactive one). Though the policy is usually the one defined in /sepolicy, you can get the actively loaded policy through `sysfs`, as well. The `/sys/fs/selinux/` directory will contain many interesting entries used for configuring (and potentially disabling) SELinux, of which one is the actively loaded policy. This will require you to do something similar to the following:

Output 21-5: Decompiling/Disassembling the active policy

```
root@htc_m8w1:/ # ls -l /sys/fs/selinux/policy /sepolicy
-r----- root      root      74982 1970-01-01 01:00 policy
-rw-r--r-- root      root      74982 1970-01-01 01:00 sepolicy
root@htc_m8w1:/ # cp /sys/fs/selinux/policy /data/local/tmp
root@htc_m8w1:/ # chmod 666 /data/local/tmp/sepolicy
#
# Back on the host (as any user)
#
morpheus@Forge (~)$ adb pull /data/local/tmp/sepolicy
2750 KB/s (74982 bytes in 0.026s)
morpheus@Forge (~)$ sedispol sepolicy
Reading policy...
libsepol.policydb_index_others: security: 1 users, 2 roles, 287 types, 1 bools
libsepol.policydb_index_others: security: 1 sens, 1024 cats
libsepol.policydb_index_others: security: 84 classes, 1333 rules, 1 cond rules
binary policy file loaded

Select a command:
1) display unconditional AVTAB
...
Command ('m' for menu): 1
allow gemud installd : udp_socket { ioctl read write create getattr setattr lock relabelfrom
  relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
  send_msg name_bind node_bind };
allow system installd : udp_socket { ioctl read write create getattr setattr lock relabelfrom
  relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
  send_msg name_bind node_bind };
allow keystore ctl_dumpstate_prop : property_service { set };
allow keystore ping : peer { recv };
... # Probably more output than your terminal can buffer - consider "f" for file output..
```

What remains, then, is to define the process of assigning the labels to resources, through **contexts**. The resources recognized by SELinux are Linux file objects (including sockets, device nodes, pipes, and other objects with a file representation), and SEAndroid extends this further to allow for properties.

Application Contexts

The `/seapp_contexts` file provides a mapping of applications (in the form of UIDs) to domains. This is used to label processes based on the UID, and the `seinfo` field (as set by the package manager, according to the package signature as it correlates with `/system/etc/security/mac_permissions.xml`). You can see the labeling of processes with the toolbox's `ps -z`:

Output 21-6: SELinux process contexts with `ps -z`

```
shell@htc_m8wl:/ $ ps -z | grep platform_app
u:r:platform_app:s0 u0_a42 7343 7129 com.android.systemui
u:r:platform_app:s0 smartcard 7717 7129 org.simalliance.openmobileapi.service:remote
u:r:platform_app:s0 u0_a42 30131 7129 com.android.systemui:recentapp
u:r:platform_app:s0 fm_radio 30405 7129 com.htc.fmsservice
#
# Compare with seapp_contexts
#
shell@htc_m8wl:/ $ grep platform_app /seapp_contexts
user=app seinfo=platform domain=platform_app type=platform_app_data_file
user=smartcard seinfo=platform domain=platform_app type=platform_app_data_file # PID 7717
user=felicarwsapp seinfo=platform domain=platform_app type=platform_app_data_file
user=irda seinfo=platform domain=platform_app type=platform_app_data_file
user=fm_radio seinfo=platform domain=platform_app type=platform_app_data_file # PID 30405
```

File Contexts

SE-Linux can associate every file with a security context. The `/file_contexts` file provides all the contexts for protected files, and the `-z` switch of toolbox's `ls` can display them, as shown in the following:

Output 21-7: SELinux file contexts with `ps -z`

```
shell@htc_m8wl:/ $ ls -Z /dev | grep video_device
drwxr-xr-x root root u:object_r:video_device:s0 video
crw-rw---- system camera u:object_r:video_device:s0 video0
crw-rw---- system camera u:object_r:video_device:s0 video1
crw-rw---- system camera u:object_r:video_device:s0 video2
crw-rw---- system camera u:object_r:video_device:s0 video3
crw-rw---- system camera u:object_r:video_device:s0 video32
crw-rw---- system camera u:object_r:video_device:s0 video33
crw-rw---- system camera u:object_r:video_device:s0 video34
crw-rw---- system camera u:object_r:video_device:s0 video35
crw-rw---- system camera u:object_r:video_device:s0 video38
crw-rw---- system camera u:object_r:video_device:s0 video39
#
# Compare with /file_contexts definitions
#
shell@htc_m8wl:/ $ grep video_device /file_contexts
/dev/nvhdcp1 u:object_r:video_device:s0 # Left over from the external/sepolicy/file_contexts,
/dev/tegra.* u:object_r:video_device:s0 # even though this is not an NVidia device
/dev/video[0-9]* u:object_r:video_device:s0 # note regular expressions gets all the above
```

Property Contexts

As discussed in [Chapter 4](#), init's property service restricts access to certain property namespaces by a hard coded uid table. This is a very rigid mechanism, and hardly scalable as new properties and namespaces are added in between Android releases.

Since SELinux already provides the notion of execution contexts, it is trivial to extend them to properties, as well. As of JellyBean, `/init` protects access to properties by a `check_mac_perms()` boolean. The function loads the property contexts from two files - `/data/security/property_contexts` (when present), and `/property_contexts`.

Output 21-8: SELinux Property contexts

```
shell@htc_m8wl:/ $ cat /property_contexts
#line 1 "external/sepolicy/property_contexts"
#####
# property service keys
#
#
net.rmnet0          u:object_r:radio_prop:s0
..
..
sys.usb.config      u:object_r:radio_prop:s0

ril.                u:object_r:rild_prop:s0

net.                u:object_r:system_prop:s0
dev.                u:object_r:system_prop:s0
runtime.            u:object_r:system_prop:s0
hw.                 u:object_r:system_prop:s0
sys.                u:object_r:system_prop:s0
sys.powerctl        u:object_r:powerctl_prop:s0
service.            u:object_r:system_prop:s0
wlan.               u:object_r:system_prop:s0
dhcp.               u:object_r:system_prop:s0
bluetooth.          u:object_r:bluetooth_prop:s0

debug.              u:object_r:shell_prop:s0
log.                 u:object_r:shell_prop:s0
service.adb.root    u:object_r:shell_prop:s0
..
```

If you go back to [Chapter 4](#), you'll see that the property contexts essentially mirror the definitions in the table. The main difference, however, is that providing the contexts in an external file provides a far more extensible way of changing and modifying properties - all without a need to recompile `/init`.

init and toolbox commands

Recall from [Chapter 4](#) that the Android `/init` has a rich variety of commands, which may be used in its `.rc` files. With the introduction of SELinux, additional commands have been added to allow for SELinux contexts. Toolbox has likewise been modified to allow SELinux modifications from the shell. Table 21-6 shows these commands

Table 21-6: init and toolbox commands for SELinux

init	Toolbox	Usage
N/A	getenforce	Get SELinux Enforcement status
setcon <i>SEcontext</i>	N/A	Set (change) SELinux context. Init uses <code>u:r:init:s0</code>
restorecon <i>path</i>	restorecon [-nrV] <i>pathname...</i>	Restore SELinux context for path
setenforce [<i>0 1</i>]	setenforce [Enforcing Permissive 1 0]	Toggle SELinux enforcement on/off
setsebool <i>name value</i>	setsebool <i>name value</i>	Toggle boolean. <i>value</i> can be 0/false/off or 1/true/on

Note you can achieve most of the functionality of the SELinux commands by accessing files in `/sys/fs/selinux` (which is, in fact, what some of these commands do), though this would require **both** root access and an unconfined domain. `/init`, which remains unconfined, can also relabel processes (as it does for services with the `seclabel` option, and additionally provides the `selinux.reload_policy` property trigger to reload the policy.

Disabling SELinux altogether can be accomplished through `/sys/fs/selinux/disable`, or through the kernel command line argument `selinux=0`.

Other noteworthy features

Linux has some additional settings which Android enables, which aim to improve security by hardening otherwise insecure defaults. This section discusses them briefly.

AT_SECURE

The Linux Kernel's ELF loader uses an auxiliary vector to provide metadata for the images it loads. This vector can be accessed through the `/proc` filesystem (as `/proc/pid/auxv`). One of its entries, `AT_SECURE` is set to a non-zero value for `set[ug]id` binaries, programs with capabilities, and programs which force an SELinux domain traversal. In those cases, Bionic's linker (`/system/bin/linker`) is configured to drop "unsafe" environment variables (a hard coded list in the `__is_unsafe_environment_variable` function, in `bionic/linker/linker_envron.cpp`). Chief amongst the variables are `LD_LIBRARY_PATH` and `LD_PRELOAD`, a favorite technique for library injection.

Address Space Layout Randomization

Code injection attacks use the target process' address space as their playing field, and their success often depends on intimate knowledge of its details - addresses, regions and protections. This is because injection attacks either directly add code into an existing program, or subvert its execution so as to jump to already existing regions. In both cases, knowledge of the layout is vital, because jumping to an incorrect address will lead to a crash. Normally, since processes enjoy their private virtual memory space and start up deterministically, a hacker can (to paraphrase an old java motto) "debug once, hack everywhere".

Address Space Layout Randomization (ASLR) attempts to make injection attacks harder by introducing randomness - shuffling the layout of memory regions, so as to make their addresses less predictable. This increases the chance a targetted piece of code will be "shifted" in memory, and basically trade a program crash in place of compromising it with malicious code - a lesser evil, by all counts.

Linux offers randomization capabilities through `/proc/sys/kernel/randomize_va_space` (or `sysctl kernel.randomize_va_space`). The value "0" specifies no randomization, "1" specifies stack randomization, and "2" specifies both stack and heap, which is the default. Executables can also be compiled with the PIE (Position-Independent-Executable) option (the `-pie` switch).



Experiment: Testing ASLR

To see the effects of ASLR, you can use the following shell script over `/proc`. The script iterates over all processes, finds the location of `libc.so` in it (only the text section, as filtered by the `grep r-x`), and displays it along with the PID if found:

Output 21-9: Showing the effects of ASLR

```
root@htc_m8wl:/# cd /proc
root@htc_m8wl:/proc # for x in [0-9]*; \
do \
    lc=`grep libc.so /proc/$x/maps | grep r-x`; \
    if [[ ! -z "$lc" ]]; then echo $lc in PID $x; fi; \
done | sort
400c6000-40111000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 28686
400f7000-40142000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 10343
..
400f7000-40142000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 9615
b6de0000-b6e2b000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 7470
b6e5a000-b6ea5000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 375
..
```

As the output shows, the library is often randomized, yet some processes still share the same location for `libc` - those are the spawns of the `zygote`, which `fork()` to load a class, but does not call `exec()` - and hence remains with the same address space layout.

Kernel-space ASLR has yet (at the time of writing) to make it into Android. Introduced for the first time in iOS 6.0, it eventually made it into Linux with version 3.14 (which is actually the most recent at the time these lines are being typed). It is quite likely to be introduced into Android with the version to follow KitKat.



The measures of ASLR discussed in this chapter provide a layer of defense only against code injected through an input vector. If an adversary already has execution privileges, s/he can invoke the powerful `ptrace(2)` APIs to read the address space of other processes, and even inject remote threads. Thankfully, in order to do so for any arbitrary processes, one has to obtain root privileges first. SELinux can (and should!) be used to prevent access to `ptrace(2)` altogether.

Kernel hardening

Unlike mainline Linux, Android kernels export no `/proc/kcore` by default, as this entry allows kernel read-only memory access from user mode (by root). The `/proc/kallsyms` is still present in most devices (and actually world readable by default), but protected by the `kernel.kptr_restrict` sysctl, which is set by default to 2, to prevent any addresses from being displayed. Kernel ring-buffer access (via the `dmesg`) is likewise protected by `kernel.dmesg_restrict`.

Stack protections

As sophisticated as attacks can get, they still (for the most part) rely on overwriting a function pointer, which - when called - causes a subversion of the program flow. Not all programs use function pointers, but all utilize the return address, which is stored on the stack during a function call.

As a countermeasure to this, most modern compilers offer automatic stack protection, by means of a *canary*. Like the proverbial canary in the coal mine, a stack canary is a random value written to the stack upon function entry, and verified right before the function returns. If the value cannot be verified, the stack is deemed corrupt, and the program voluntarily aborts, rather than potential trigger malicious code.

This form of protection has been available in Android since its early days, with gcc's `-fstack-protector`. Note that it does not provide a panacea, since code can still be injected via function pointers aside from the return address (C++ methods make good candidates).

Data Execution Prevention

Code injection attacks rely on embedding malicious code inside input - whether direct from the user or from other sources. Input, however, is data - and memory used for data (the heap and the stack) can be flagged as non-executable. This complicates attacks somewhat, because just using the classic trampoline technique (overwriting a pointer or the stack return address with the address of the injected code) won't work if the injected code is in the data segment.

Unfortunately(?) while making data non-executable complicates the simple attacks, attacks have considerably evolved. The current counterattack is Return-Oriented-Programming (ROP), a fairly old technique (introduced by Solar Designer in a '00 paper as *return-to-libc*), which strings together "gadgets" of calls back into existing portions of code in the program, simulating function calls on the stack. Because these are calls into code, there's nothing to make non-executable, and thus the protection can be fairly reliably circumvented.

Compiler-level protections

All the above protections are, in a way, treating the symptoms, rather than the disease. At the end of the day, the only proper ways to combat code injection attacks which exploit memory corruption is to exercise defensive coding, which involves input validation and strict bounds checking on memory operations. Newer versions of Android have taken that to heart, with the source compiled with enhanced checks, most notably `FORTIFY_SOURCE` and `-Wformat-security`, which add additional checks on memory copying functions, and prevent format string attacks.

Security at the Dalvik Level

Dalvik Level Permissions

Working at the level of a virtual machine, rather than native code, brings with it tremendous advantages for monitoring operations and enforcing security. At the native level, one would have to monitor system calls for any significant resource access. The problem with system calls, however, is that their granularity is inaccurate. File access is straightforward (open/read/write/close), but other operations, (e.g. a DNS lookup) are a lot harder to monitor, as they involve multiple system calls. Therein lies the advantage of the Virtual Machine - most operations are carried out by means of pre-supplied packages and classes, and those come built-in with permission checks.

Android actually takes this a step further: Whereas in a normal Java class a malicious developer could ostensibly import other classes, implement functionality from scratch or use JNI (to break out of the VM), in order to avoid permission checks, though this is next to impossible in Android: The user application is entirely powerless, devoid of all capabilities and permissions at the Linux level, so any access to the underlying system resources should be blocked right there. In order to carry out any operation which has an effect outside the scope of the application, one has to involve `system_server`, by calling `getSystemService()`.

While any app can freely invoke a call to `system_server`, none has access to its defined permissions - which `system_server` will check. This check is performed outside the application's process, so the application has no plausible avenue by means of which it may somehow obtain those permissions, unless they were a priori assigned to it. The assignment is performed when the application is loaded and installed - meaning that the user has been notified of the application's requested permissions, (has hopefully read through the very long list), and approved them (again, hopefully knowing the ramifications of hitting "OK"). If the permission requested during runtime has been revoked (for example, through the AppOps service or through `pm revoke`), a security exception will be thrown (normally, this will crash the application, unless the developer braced for such an exception, in which case it may handle the exception, usually popping up an explanation on what permission was required, or at other times failing silently).

What follows is that the permissions themselves need no special data structures or complicated metadata. A permission in Dalvik is nothing more than a simple constant value, which is granted to an application in its manifest, as it declares it `uses-permission`. An application can likewise define its own constants (as `permission` tags in the Manifest). The permission enforcement is performed by the Package Manager, and you can use the `pm` command (among other things) to display and control many aspects of permissions, as shown in the following experiment:



Experiment: Using the `pm` command

You can use `pm list permissions` to display permissions, both of the Android frameworks, and of third party applications. To do so, try:

Output 21-10: Listing permissions with `pm`

```
root@htc_m8wl:/# pm list permissions -f | more
All Permissions:
+ permission:android.permission.GET_TOP_ACTIVITY_INFO
  package:android
  label:get current app info
  description:Allows the holder to retrieve private information about the current application
  protectionLevel:signature
.. # Application declared permissions, as imported from their AndroidManifest.xml
+ permission:com.facebook.system.permission.READ_NOTIFICATIONS
  package:android
  label:null
  description:null
  protectionLevel:signature
..
```

Other useful switches include `-s` (verbose human readable output in your locale), `-g` (permission groups). The `pm` command can also be used to grant and revoke optional permissions (`pm [grant|revoke] PACKAGE PERMISSIONS`) and even toggle permission enforcement (i.e. `pm set-permission-enforced PERMISSION [true|false]`). The full syntax of this command is explained in [Chapter 9](#).

The AppOps service (detailed in [Chapter 5](#)) provided a powerful GUI by means of which users could track and fine-grain tune application permission usage. The GUI has been removed as part of KitKat's 4.4.2 "security update", though the functionality remains.

Dalvik Code Signing

Permissions by themselves are somewhat useless - after all, any app can declare whatever permissions it requires in its `AndroidManifest.xml`, and the unwitting user will probably click "ok" when prompted. To bolster security, Google requires digital signatures on applications uploaded to the Play store, so as to identify the developer(s) behind them, and add accountability.

Thus, all Android applications must be signed (with the process explained in [Chapter 6](#). What's not so clear is - by whom. As Google was playing catch-up to Apple and opened the Play Store, it wanted to offer an advantage to developers, in the form of a simpler process. As opposed to Apple's lengthy validation process - all apps must be vetted by Apple, and digitally signed by them, Google offered anyone the ability to just create a key pair, publish their public key, and use the private key to sign their APK file. The rationale was that this achieves a similar level of identifying the APK's source, while at the same time greatly simplifying the process of submitting applications to the Store.

In practice, this led to an explosion of Malware in the Play Store. The Google approach was that any malware found and reported would be removed from the Store, and the corresponding public keys blacklisted. From the malware author's side, this was a case of "better to beg forgiveness than ask permission" - as the malware by then would have likely propagated by the time it was detected, thus achieving its purpose. This, coupled with the fact that a malware developer could always generate a new key pair, hollowed out the entire security model. A [recent study published in RSA 2014](#) found that "malicious apps have grown 388 percent from 2011 to 2013, while the number of malicious apps removed annually by Google has dropped from 60% in 2011 to 23% in 2013", and that effectively one out of every 8 apps in the store is, in fact, malicious.

The Android "Master Key" vulnerability

One of the most serious vulnerabilities discovered in Android (in 2013) is what came to be known (somewhat erroneously) as the "Master Key Vulnerability". The vulnerability (discovered by [BlueBox security](#), and refined (among others) by [Saurik](#), the noted iOS creator of Cydia) occurred in of mishandling of APK files which contained files with duplicate names. APKs are ZIP files, and normally most utilities - aapt included - would not allow duplicate file names in the same zip. Technically, however, it *is* possible, and introduced a peculiar vulnerability: File signature verification was performed on the first entry in the APK, whereas extraction was performed on the second! This oddity was due to two different libraries - Java's and Dalvik's native implementation - being used for the tasks. As a consequence, it followed that anyone could take a validly signed APK file, and just add additional files with the same names as the original (including `classes.dex`, of course). This effectively bypassed Android's signature validation on APKs. Though fixed, the bug is a great example of oftentimes gaping vulnerabilities which need little to no technical knowledge in order to exploit.

The Intent Firewall

The Intent Firewall is - as its name implies - an extra measure of security when processing intents, allowing for an additional rulebase used for filtering, in a manner akin to a network firewall blocking certain ports. The feature has been added in 4.3, though remains largely unused at the time of writing.

The Intent Firewall is implemented by its eponymous class, which looks for its configuration files in `/data/system/ifw`. If configuration files are found (in practice, this directory is empty in most devices), they are loaded. The rule files must be suffixed as `.xml`, and contain a `<rules>` element, under which may be rule tags for `<activity>`, `<service>` and `<broadcast>`. The class also adds a `FileObserver` on the directory, to automatically load or remove new rules as the directory contents are modified.

Other than that, the Firewall remains passive, and waits to be called. The Activity Manager is its main client, calling the firewall methods - `checkStartActivity`, `checkService` and `checkBroadcast`. Those internally all call `checkIntent`, which check the rulebase, potentially log, and return a boolean affirmation or denial.

User Level Security

So far, the discussion in this chapter focused on application level security. Android also needs to offer security at the user-level, allowing only the legitimate device user access to it, and in particular its sensitive data. Beginning with JellyBean, Android supports multiple users, which complicates matters a little.

The Lock Screen

The lock screen is a device's first and only real line of defense against theft or physical interception by malicious entities. It is also the screen most often seen by the user, when the device awakens from its frequent slumber. As such, it must be made resilient, on the one hand, but also natural and quick, on the other. As with most Android features, vendors may customize this screen, though Android provides an implementation which is often used as is.

Passwords, PINs and Patterns

The default Android lock screen allows either passwords, PINs or "patterns". Patterns are, in effect, PINs, but instead of remembering actual digits, the user simply has to swipe a grid (usually 3x3). The user can opt for an actual PIN instead, which is technically stronger than a pattern in that its length may be unlimited, and it may repeat digits. A password provides a further enhancement over a PIN in that it allows a mix of different case letters and numbers.

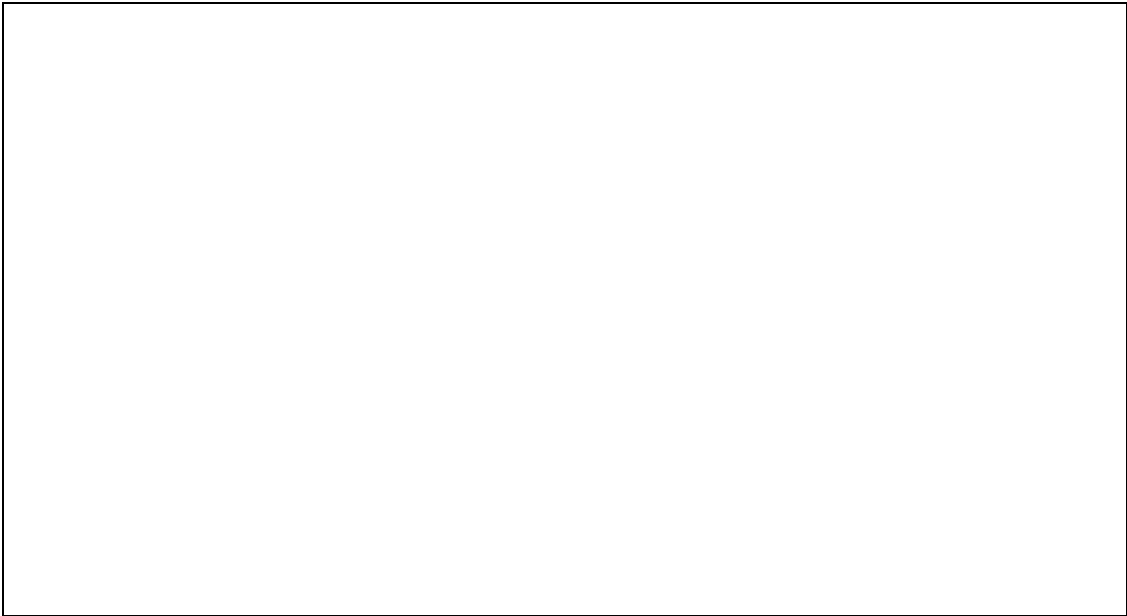
The lock screen is, in effect, just an activity, implemented the `com.android.keyguard` package. The package contains all the primitives for the system supplied lock screens and methods, and includes the following classes:

Table 21-7: The classes in `com.android.keyguard`

Class	provides
BiometricSensorUnlock	Interface used for biometric methods, e.g. FaceUnlock
Keyguard[PIN SimPin Password]View	Default views to prompt for PIN or password credentials
KeyguardSecurityView	Interface implemented by keyguard views (emulates activity lifecycle)
KeyguardService	Keyguard Service implementation
KeyguardSecurityCallback	Interface implemented by KeyguardHostView
KeyguardViewMediator	Mediates events to the Keyguard view

The lock screen invocation begins when the power manager wakes up the display, and notifies the implementation of the `WindowPolicyManager`. This calls the `KeyguardServiceDelegate`'s `onScreenTurnedOn`, which waits for the `keyGuard`. From there, it falls on the `keyGuard` to draw the lock screen (via some activity), and handle whatever lock credentials mechanism was chosen by the user. The lock screen can also be invoked from the `DevicePolicyManager`'s `lockNow` method, as shown in figure 21-1:

Figure 21-1: Starting the lock screen



The actual logic of handling the lock is performed by `LockPatternUtils`, which calls on the `LockSettingsService`, a thread of `system_server`. The service, in turn, verifies the input against `LOCK_PATTERN_FILE` (gesture.key) or `LOCK_PASSWORD_FILE` (password.key, for PINs and passwords alike). In both cases, neither pattern nor passwords are actually saved in the file, but their hashes are. The service additionally uses the `locksettings.db` file, which is a SQLite database which holds the various settings for the lock screen. Those are shown in table 21-8:

Table 21-8: The `locksettings.db` database keys

LockPatternUtils constant	Key name (lockscreen.*)
<code>LOCKOUT_PERMANENT_KEY</code>	<code>lockedoutpermanently</code>
<code>LOCKOUT_ATTEMPT_DEADLINE</code>	<code>lockedoutattempteddeadline</code>
<code>PATTERN_EVER_CHOSEN_KEY</code>	<code>patterneverchosen</code>
<code>PASSWORD_TYPE_KEY</code>	<code>password_type</code>
<code>PASSWORD_TYPE_ALTERNATE_KEY</code>	<code>password_type_alternate</code>
<code>PASSWORD_TYPE_ALTERNATE_KEY</code>	<code>password_type_alternate</code>
<code>LOCK_PASSWORD_SALT_KEY</code>	<code>password_salt</code>
<code>DISABLE_LOCKSCREEN_KEY</code>	<code>disabled</code>
<code>LOCKSCREEN_BIOMETRIC_WEAK_FALLBACK</code>	<code>biometric_weak_fallback</code>
<code>BIOMETRIC_WEAK_EVER_CHOSEN_KEY</code>	<code>biometricweakeverchosen</code>
<code>LOCKSCREEN_POWER_BUTTON_INSTANTLY_LOCKS</code>	<code>power_button_instantly_locks</code>
<code>LOCKSCREEN_WIDGETS_ENABLED</code>	<code>widgets_enabled</code>
<code>PASSWORD_HISTORY_KEY</code>	<code>passwordhistory</code>



Experiment: Viewing the `locksettings.db`

If your device is rooted and you have the SQLite3 binary installed, you can inspect the `locksettings.db` file. You can also use `adb` to pull the `locksettings.db` to your host.

Output 21-11 Viewing the lock settings Database

```
root@htc_m8wl:/data # sqlite3 /data/system/locksettings.db
SQLite version 3.7.11 2012-03-20 11:35:50
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE android_metadata (locale TEXT);
INSERT INTO "android_metadata" VALUES('en_US');
CREATE TABLE locksettings (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                           name TEXT,user INTEGER,value TEXT);
INSERT INTO locksettings VALUES(2,'lockscreen.options',0,'enable_facelock');
INSERT INTO locksettings VALUES(3,'migrated',0,'true');
INSERT INTO locksettings VALUES(4,'lock_screen_owner_info_enabled',0,'0');
INSERT INTO locksettings VALUES(5,'migrated_user_specific',0,'true');
INSERT INTO locksettings VALUES(9,'lockscreen.patterneverchosen',0,'1');
INSERT INTO locksettings VALUES(11,'lock_pattern_visible_pattern',0,'1');
INSERT INTO locksettings VALUES(12,'lockscreen.password_salt',0,'-3846188034160474427');
INSERT INTO locksettings VALUES(81,'lockscreen.disabled',0,'1');          # No Lock
INSERT INTO locksettings VALUES(82,'lock_fingerprint_autolock',0,'0');
INSERT INTO locksettings VALUES(83,'lockscreen.alternate_method',0,'0');
INSERT INTO locksettings VALUES(84,'lock_pattern_autolock',0,'0');
INSERT INTO locksettings VALUES(86,'lockscreen.password_type_alternate',0,'0');
INSERT INTO locksettings VALUES(87,'lockscreen.password_type',0,'131072');  # PIN
INSERT INTO locksettings VALUES(88,'lockscreen.passwordhistory',0,'');
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('locksettings',88);
COMMIT;
sqlite>
```

The columns in the `locksettings` table includes "user" (to support Android Multi-User login, as of JB). The values are usually boolean (0/1), but not always - there are some flag combinations, and a salt for the .key file. You can use SQL statements to change the lock settings from within SQLite3. though they will be cached by the lock settings service. You can also just rename the file - if you do so and restart `system_server`, it will be recreated with the defaults (and also have the nice side effect of resetting your password or pattern).

Alternate lock methods

Ice Cream Sandwich introduced face recognition as an alternative to the traditional methods. This was touted to much fanfare, as a potential differentiator against iOS. Unfortunately, the recognition rates are far from perfect - figures range from as low as 60% to 90%. Face recognition can also easily be defeated - by holding up a picture to the phone. Ironically, people who have tried this method found it works with greater accuracy than the user's actual face...

The Motorola Atrix 4G was the first Android device to implement fingerprint scanning as an alternative method. This also suffered poor recognition rates. Apple's acquisition of AuthenTec in 2012 suggested fingerprint authentication was coming to iOS and, indeed, it made its debut in the iPhone 5S. Samsung initially slammed this as a poor, uninnovative feature, but nonetheless (and unsurprisingly) went on to introduce it to their "next big thing", the Galaxy S5. Other Android vendors are quickly following, and it seems this will become a standard feature.

Multi-User Support

For the majority of its existence, Android has operated under the assumption that the device only has one user. Unlike desktop systems, which have long allowed user login and switching, this feature was only introduced into Android with JellyBean (4.2), and has been initially introduced only into tablets.

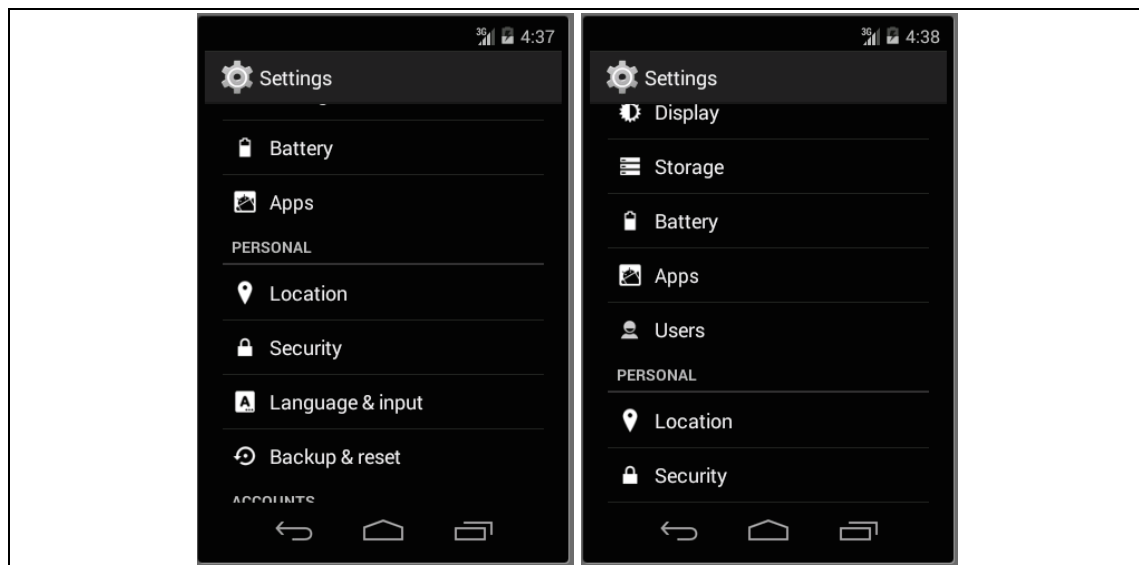
Android already uses the user IDs for the individual applications, as explained previously. To implement multi-user support, it builds on the same concept, by carving up the AID space into non-overlapping regions, and allocating one of every human user. Application IDs are thus renamed from `app_###` to `u##_a###`, and users are created with separate directories in `/data/user`. Application data directories are moved to `/data/user/##/`, with the primary user being user "0". The legacy `/data/data` thus becomes the primary user's directory (symlinked from `/data/user/0`). The user profiles themselves are stored in `/data/system/users`. This is shown in the following experiment:



Experiment: Enabling multi-user support on API 17 and later

On tablets, multi-user support will be enabled by default as of JellyBean (API 17). A little known feature, however, is that you can enable it on phones as well. All it takes is setting a system property - `fw.max_users` to any value greater than 1. Doing so on the Android emulator will bring up the "Users" option to settings, as shown in the following screenshot:

Screenshot 21-1: Before and After `fw.max_users` property modification





Experiment: Enabling multi-user support on API 17 and later (cont)

Adding a user is straightforward, though the system will force you to set a lock screen, in order to differentiate between the two users on login. The process should look something like this:

Output 21-12: Listing multiple user profiles in /data/system/users

```
root@generic:/data/system/users # ls -F
0/
0.xml
userlist.xml
root@generic:/data/system/users # setprop fw.max_users 3
#
# Add another user through settings.. (shell stop/start might be necessary) then ls again
#
root@generic:/data/system/users # ls -F
0/
0.xml
10/
10.xml
userlist.xml
root@generic:/data/system/users # cat userlist.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<users nextSerialNumber="11" version="4">
  <user id="0" />
  <user id="10" />
</users>
root@generic:/data/system/users # cat 0.xml # Display details for user 0
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<user id="0" serialNumber="0" flags="19" created="0" lastLoggedIn="1400702272027"
  icon="/data/system/users/0/photo.png">
  <name>Owner.com</name>
  <restrictions /> # restrictions, if any, go in this element
</user>
root@generic:/data/system/users # ls -l 0 # Show settings for user 0
-rw-rw---- system system 196608 2014-05-21 20:40 accounts.db
-rw----- system system 33344 2014-05-21 20:40 accounts.db-journal
-rw----- system system 944 2014-05-21 21:17 appwidgets.xml
-rw-rw---- system system 17970 2014-05-21 21:13 package-restrictions.xml
-rw----- system system 357887 2014-03-28 16:24 photo.png
-rwx----- system system 1896839 2014-04-04 01:05 wallpaper
-rw----- system system 99 2014-05-21 04:39 wallpaper_info.xml
```

From the command line, the same effect can be achieved by using `pm create-user`, which connects to the user manager (`IUserManager.Stub.asInterface(ServiceManager.getService("user"))`) and invokes its `createUser()` method. The `pm remove-user` command likewise removes a given user.

Depending on how you create the user (as a separate user or a restricted user, which shares the original user's apps), the `restrictions` element may be populated with the following boolean attributes, all defined in the `android.os.UserManager` class. The actual handling of the files (above) and the restrictions is performed by `com.android.server.pm.UserManagerService`.

Table 21-9: User Restrictions

Restriction
<code>no_modify_accounts</code>
<code>no_config_wifi</code>
<code>no_install_apps</code>
<code>no_uninstall_apps</code>
<code>no_share_location</code>
<code>no_install_unknown_sources</code>
<code>no_config_bluetooth</code>
<code>no_config_credentials</code>
<code>no_remote_user</code>

Key Management

Android relies extensively on cryptographic keys, for system internal use (validating installed packages) and for application use. In both cases, the keystore service (discussed in [Chapter 4](#)) plays an integral part in abstracting and hiding the implementation.

Certificate Management

Public Key Infrastructure is the de-facto fulcrum of all Internet security. Encryption rests on several key assumptions which relate to the algorithms and methods behind public keys, the most important of which is a *trust*. Simply put, this means that if you know a subject's public key, the key can be used not just for encrypting messages to it, but also authenticating messages from it. This, in turn, means that if this subject vouches for another public key by authenticating it, (which is, in effect, what a certificate is). In this way, a *trust hierarchy* can be formed.

This principle, while powerful, does lead to a chicken and egg problem - you can authenticate a public key only if some other public key has been a priori used to authenticate it. The way around this predicament is to hard code the initial public keys in the operating system. These keys are encoded in the form of *root certificates* - public keys authenticating themselves. When passed over the network, they are of no value (as they are trivial to spoof). When hard-coded, however, they can be trusted and provide the basis for the trust hierarchy.

Android hard-codes root certificates in `/system/etc/security/cacerts`. The certificates are encoded in their PEM (Privacy-Enhanced-Mail) form, which is a Base64 encoding of the certificate between delimiters. Some devices will also have the plain ASCII form of the certificate before or after the PEM encoding. If not, it's a simple matter to display it using the `openssl` command line utility, which is built-in to Linux or Mac OS, shown in output 21-13:

Output 21-13: Using `openssl` to decode a PEM certificate

```
morpheus@Forge (/tmp)$ adb pull /system/etc/security/cacerts
pull: building file list...
pull: /system/etc/security/cacerts/ff783690.0 -> ./cacerts/ff783690.0
..
morpheus@Forge (/tmp)$ openssl x509 -in ff783690.0 -text | more
Certificate:
  Data:
    Version: 3 (0x2)      # Denotes the X.509v3 format
    Serial Number:        # Used to refer to certificate when revoking
                          44:be:0c:8b:50:00:24:b4:11:d3:36:2a:fe:65:0a:fd
    Signature Algorithm:  sha1WithRSAEncryption
    Issuer: # Issuer in LDAP notation: C=country, ST=state, L=location,
              # O=Organization, OU= Organizational Unit, CN=Common Name
    Validity
      Not Before: # Usually coincides with certificate issue date
      Not After : # Usually set to 2-10 of years from issue date
    Subject: # Certificate Owner, in same LDAP notation
              # ...
    Subject Public Key Info:
      .. # Modulus and Exponent (usually 65537)
    X509v3 extensions:
      X509v3 Key Usage:
        Digital Signature, Non Repudiation, Certificate Sign, CRL Sign
      X509v3 Basic Constraints: critical
        CA:TRUE
      X509v3 Subject Key Identifier:
        A1:72:5F:26:1B:28:98:43:95:5D:07:37:D5:85:96:9D:4B:D2:C3:45
      X509v3 CRL Distribution Points:
        URI:http://crl.usertrust.com/UTN-USERFirst-Hardware.crl
      X509v3 Extended Key Usage:
        TLS Web Server Authentication, IPSec End System, IPSec Tunnel, IPSec User
    Signature Algorithm: sha1WithRSAEncryption
      # .. SHA-1 hash of certificate, signed with RSA private of issuer
      # ..
-----BEGIN CERTIFICATE-----
MIIEEdCCAlYgAwIBAgIQRL4M1AAJLQR0zYq/mUK/TANBgkqhkiG9w0BAQUFADCB
lzELMAkGA1UEBhMCVVMxCzAJBgNVBAGTA1VUMRcwFQYDVQQHEw5TYWx0IEExha2Ug
... Base 64 (original PEM) encoding of the certificate
KqMiDP+JJnlfIytH1xUdgWqeUQ0qUZ6B+dQ7XnASfxAynB67nfhmQA==
-----END CERTIFICATE-----
```

Of special importance are the Over-The-Air (OTA) update certificates, stored in the `/system/etc/security/otacerts.zip` archive. The archive usually contains one (rarely, more) certificates which are used for validating OTA updates (described in [Chapter 3](#)). The `RecoverySystem` class parses this file (hardcoded as `DEFAULT_KEYSTORE`), in its `getTrustedCerts()` method using a `CertificateFactory`. Once again, any certificates would be encoded in PEM (usually, without human readable text), but you can use the method shown in output 21-13 to decode them. Removing this file is a good method to "combat" auto-updates in some Android distributions (such as FireOS), which may cause you to lose root access post-update.

Certificate Pinning

JellyBean (API 17) introduces *certificate pinning*, which has become a common add-on to SSL certificate validation. Pinning involves hard-coding the expected public key of a host (via its certificate), so that if the host presents a certificate which does not match the pin (or one of the pins in a pin set) it is rejected.

Unlike the certificates discussed previously, which are in `/system/etc/security` (and therefore cannot be modified), pins are maintained in `/data/misc/keychain/pins`, which is a file that can be replaced. The `CertPinInstallReceiver` class registers a broadcast receiver for the `UPDATE_PINS` intent, and - when such an intent is received, its extras are expected to contain the following:

- **`EXTRA_CONTENT_PATH`**: The file name containing the new pins.
- **`EXTRA_VERSION_NUMBER`**: Which is expected to be greater than the current version.
- **`EXTRA_REQUIRED_HASH`**: Of the current pins file.
- **`EXTRA_SIGNATURE`**: Signature of the file supplied, its version and hash of current pins file

The `CertPinInstallReceiver`'s `onReceive` (inherited from `ConfigUpdateInstallerReceiver`) gets the values from the broadcast intent, ensures the version number is indeed greater than the current version of the pins file (in `/data/misc/keychain/metadata/version`), and that the current file's hash matches the hash specified in the intent. It then verifies the signature, using the certificate stored in the system settings database under `config_update_certificate` (the `UPDATE_CERTIFICATE_KEY`). If everything is in order, the filename from the intent is copied over the existing pins file, and the `metadata/version` is updated to reflect the new version number.

Google pins all of its (many) certificates by default, and the vendor may pin additional ones. A quick way of looking at pins is shown in Output 21-14:

Output 21-14: Displaying the pinned domains

```
root@htc_m8w1:/ # cat /data/misc/keychain/pins | cut -d"=" -f1
*.spreadsheets.google.com
*.chart.apis.google.com
appengine.google.com
*.google-analytics.com
*.doubleclick.net
*.chrome.google.com
*.googleapis.com
*.apis.google.com
*.drive.google.com
*.checkout.google.com
www.gmail.com
*.groups.google.com
*.ytimg.com
googlemail.com
*.sites.google.com
*.google.com
*.googleusercontent.com
*.googleadservices.com
*.youtube.com
*.profiles.google.com
*.accounts.google.com
*.ssl.gstatic.com
*.appspot.com
*.mail.google.com
www.googlemail.com
*.plus.google.com # largely unused ;-)
gmail.com
*.health.google.com
..
```

The [Android Explorations Blog](#) contains a sample application demonstrating the creation of a pins file and its update operation through the intent.

Certificate Blacklisting

Android provides the `CertBlacklister` class to handle black listing (effectively, revocation) of certificates. The class (instantiated as a service of `system_server`, as discussed in [Chapter 5](#)) registers an observer for two content URIs:

- `content://settings/secure/pubkey_blacklist`: Stores known compromised or revoked public keys or certificates. Content written here ends up written to `/data/misc/keychain/pubkey_blacklist.txt`.
- `content://settings/secure/serial_blacklist`: Stores known compromised or revoked serial numbers of certificates. Serial numbers written here are saved to `/data/misc/keychain/serial_blacklist.txt`.

Both values are also in the system's secure settings, as can be seen in the following output:

Output 21-15: Viewing the serial and pubkey blacklists

```
root@htc_m8wl:/ # sqlite3 /data/data/com.android.providers.settings/databases/settings.db \
"select * from secure" | grep black
95|serial_blacklist|827,864
99|pubkey_blacklist|5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa..
root@htc_m8wl:/ # cat /data/misc/keychain/serial_blacklist.txt
827,864
root@htc_m8wl:/ # cat /data/misc/keychain/pubkey_blacklist.txt
5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa9101913e8e
```

Secret and Private Key Management

Storing secrets - symmetric keys or the private part of a public-key pair - poses serious challenges for any security infrastructure. If one assumes that file permissions are a strong enough layer of security, the secrets can be placed in a file and appropriately locked down. The underlying file permissions of Linux, however, are inflexible, and configuration errors could lead to secret leakage. Likewise, there is the problem of obtaining root access - which effectively voids all permissions, leaving everything in the clear.

Android provides access to secrets via the `keystore` service. This service has already been discussed in [Chapter 4](#). Keystores for applications are maintained on a per-user basis, in the `/data/misc/keystore/user_##` directory, but applications have no direct access to that directory, and must go through the keystore service, which is the sole owner of the directory (permissions 0700). The service also provides public key functions - `generate`, `sign` and `verify` - without allowing applications any access to the underlying private keys. This allows the key storage to be potentially implemented in hardware.

Indeed, Android offers hardware backed secure storage, on those devices which support it, as of JellyBean. As discussed in [Chapter 11](#), the `keymaster` HAL abstraction provides both a uniform interface for encryption operations, and allows its implementation in both software and hardware. Thus, supporting devices implement a hardware backed keymaster module, whereas those which do not use a `softkeymaster` instead.

/data Encryption

While most users remain oblivious to the need for encryption on their devices, corporate users certainly fear the compromising of data which would ensue should a device be lost or stolen. iOS provided transparent encryption as of iOS 4, and coincidentally, so has Android as of Honeycomb. By using the very same dm-crypt mechanism utilized by OBBs and ASEC, Honeycomb extends the notion of encryption to the full filesystem layer. The term "full disk encryption" is therefore somewhat inaccurate here, since it is only the /data partition which is normally encrypted. This actually makes more sense, because /system contains no sensitive data (and would be impacted from the latency incurred by crypto-operations).

Android's documentation [provides a detailed explanation of encryption](#). As with ASECs and OBBs, the volume manager is responsible for performing both the filesystem encryption and decryption. The former is performed when selected by the user, and is a rather lengthy operation. The latter is performed transparently, when the encrypted filesystem is mounted as a block device using the device mapper.

Note, that unlike [obb](#) and [asec](#) - the decryption keys for which are stashed somewhere on the device in plaintext, albeit readable only by root - the key for the /data partition encryption does not actually reside on the device, but requires the user to interact during boot, and supply it (or, more accurately, the password from which this key is derived). This requires modifications to the Android boot process, as well as an interaction between init and vold, which we describe in [Chapter 4](#).

Prior to the dm-crypt solution, there were several proposed alternatives for file system encryption (most notably [EncFS by Wang et Al.](#)), but the dm-crypt one is likely to remain as the de facto standard, even if it not yet enabled by default.

Secure Boot

KitKat introduced a new feature for securing the boot process, using the kernel's device mapper. This feature, known as **dm-verity** originated in Chromium OS, and has been ported into Linux (and thus Android), beginning with kernel version 3.4.

Recall from [Chapter 3](#), that a chain of trust (also known as the *verified boot path*) has been established from the ROM, via the boot loader, and onto the kernel and the root file system (i.e. the boot partition). While the bootloader actually does verify /system, it does so only when flashing the entire partition - which leaves open the avenue for a root owned process (be it "rooting" or malware) to make persistent changes in /system, by remounting it as read-write, and modifying files in it. Using dm-verity effectively extends the boot chain of trust one more level, onto /system.

Verifying the integrity of a partition is the simple matter of hashing all of its blocks (DM-Verity uses SHA-256), and comparing that hash against a stored, digitally signed hash value. To do so effectively, however, one has to avoid the lengthy process of reading the entire partition, which can delay boot. To get around this limitation, dm-verity reads the entire partition only once, and records the hash value of each 4k block in the leaf nodes of the tree. Multiple leaf nodes are rehashed in the second level of the tree, and then onward to the third, until a single hash value is calculated for the entire partition - this is known as the **root hash**. This hash is digitally signed with the vendor's private key, and can be verified with its public key. Since disk operations are performed in full blocks, it is a straightforward to add an additional hash verification on the block as it is placed into the kernel's buffer/page cache, and before it is returned to the requester. If the hash check fails, an I/O error occurs, and the block is known to be corrupted.

The dm-verity feature is touted for malware prevention, since it effectively prevents any modification of /system, but does have the side effect of preventing unauthorized persistent rooting, as well. Malware could definitely attempt to make modifications to /system, but Android would detect them, potentially refusing to boot - yet the same would apply for any "persistent root" back door, e.g. dropping a SetUID /system/sbin/su. From the vendor's perspective, this is fine - most vendors would only provide root via bootloader unlocking, which breaks the chain of trust at its very first link. Further, dm-verity requires only a subtle modification to the update process (discussed in [Chapter 3](#)) - namely, that the vendor regenerate the signature when /system is modified during an update. Otherwise, /system remains read only throughout the device's lifetime, and the signature must therefore remain intact.

The kernel mode implementation of dm-verity is rather small - a 20k file of drivers/dm/dm-verity.c, which plugs into the Linux Device Mapper (as discussed in [Chapter 17](#)). Google details the verified boot process in the [Android Documentation](#). The [Android Explorations Blog](#) provides further detail, including using the `veritysetup` during the building of the image.

Rooting Android

Most vendors provide ADB functionality on their devices and leave the operating system relatively open for developers, but few (if any) provide root access to the device. There is a strong rationale not to do so, considering that obtaining root access to a UNIX system brings with it virtual omnipotence - and Android is no different. Leaving behind open access to root would also potentially provide an attack vector for malware (which Android knows no shortage of). With root access, any file on the system could be read, or - worse - overwritten, which would give an attacker both access to private data, as well as the ability to hijack control of the device.

The same can be said for Apple's iOS (also a UNIX system, based on Darwin), but herein lies the significant difference between the two. Apple's developers have engineered the system from the ground up, literally, from the very hardware to the uppermost layers of software, to be rock solid and not to allow root access (in fact, not to allow *any* access aside from a sandboxed app model) at all costs. Android is built on Linux, which itself is an amalgam of code strains from different contributors, not all of which adhere to the strictest security standards. Additionally, several vendors leave an avenue, which can be exploited (by a human user in possession of the device) to gain root access - redirecting the system to boot an alternate configuration. Another way of looking at it is, Android considers the application to be the enemy - whereas iOS considers the user itself to be one.

Boot-To-Root

When Android devices boot, they normally do so by the process described in [Chapter 3](#). It is possible, however, to divert the boot process to an alternate boot, for "safe" boot, system upgrade, or recovery. This can usually be done by pressing a physical button combination (usually one or both of the volume buttons, and the home button, if it exists), or by a fastboot command, when the device is connected over USB. Once the boot flow is diverted, the boot loader can be directed to load an alternate boot image - either the on-flash recovery image, an update supplied on the SD-card, or (over USB) an image supplied through fastboot.

If a device's bootloader can be unlocked (as explained in [Chapter 3](#)) the device can be rooted. It's that simple. As previously mentioned, unlocking the boot loader will cause /data to be effaced, in an effort to prevent the user's sensitive data from falling into the wrong hands. Additionally, some boot loaders will permanently set a flag indicating that the loader has been tampered with, even if it is re-locked at some point. This is to note that the boot loader basically shirks all responsibility for system security, as it will no longer enforce digital signatures on images flashed.

All it takes to "root" the device is really just one part of the device image - the init RAM disk (initramfs). Because the kernel mounts the initrd as the root filesystem and starts its /init with root privileges, supplying an alternate /init - or even just a different /init.rc file - suffices to obtain root access. From that point onwards, it's a simple matter of convenience: It's straightforward to simply have ADB maintain root privileges (by setting `ro.secure=0`) or replace adb to a version which doesn't drop privileges. Most rooting tools, however, usually drop a `su` binary into /system/bin or /system/xbin, and use `chmod 4755` to toggle the setuid bit, so when it is invoked from the shell, the setuid effect will kick in, and automatically bestow root permissions. The code for such a binary (pre Kit-Kat) is so simple it can be summarized in three functional lines:

Listing 21-5: A simple implementation of `su`, for non SE-Linux enforced devices

```
#include <stdio.h>
void main(int argc, char **argv)
{
    setuid(0);
    setgid(0);
    system("/system/bin/sh");
}
```

You can find a similar implementation (with command line options) in the AOSP's /system/extras/su/su.c. As of KitKat, however, the introduction of SE-Linux in enforcing mode makes the binary less trivial, in that its parent (the shell) is already confined to a restricted execution context (`u:r:shell:s0`), which it cannot break out of. This requires the `su` binary to make an IPC call to a process in the `u:r:kernel:s0` unrestricted context, to then spawn a shell (e.g. the WeakSauce exploit (with DaemonSu), as explained on the [book's companion website](#)).

The practice of rooting is so popular that there are quite a few "SuperUser" applications, which provide a GUI interface to administer root access, once the device is rooted. The applications actually offer a programmatic API (via permissions and intents) to allow other applications access to root. One notable example is chainfire's SuperSU, which defines its own Dalvik level permissions

(`android.permission.ACCESS_SUPERUSER` and `eu.chainfire.supersu.permission.NATIVE`) and enables applications to broadcast intents in order to obtain superuser privileges.

Rooting via Exploiting

Whether or not a vendor has left the boot-root backdoor open, often there exist additional backdoors. These, unlike the former, are quite unintentional, and all rely on some form of system vulnerability exploitation. The ways to do so are myriad, and often unpredictable until discovered, but they all share the same common denominator: Find some insecure configuration setting or software component, and trigger some code path, by means of which root access can be obtained. The security jargon for these attack types is **privilege escalation**, as it refers to the process wherein a lower privilege process (that is, some app), can increase its privileges, usually first to those of the system user, and then root.

There is a strong similarity between exploit-based rooting methods and "jailbreaking" for iOS. In both cases, it takes the discovery and exploitation of software bugs, and both methods *should not* be possible in a perfect world (at least, according to Google and Apple). Once these methods are discovered, their days are numbered: The operating system is fairly quickly patched, and suggested to the user for download and updated (or even auto-updated, as for example with the Amazon Kindle). One prominent example was in Gingerbread, wherein Google itself pushed an update for a vulnerability in the Linux kernel, known at the time to have been actively exploited by malware.

A thorough discussion of exploitation techniques is thus beyond the scope of this work, and quite frankly is pointless, since all known exploits at this time have been patched. Exploits generally obtain root by passing crafted input to a process already running as root (vold has been a perennial favorite..), corrupting its memory (stack or heap) and usually overwriting a function pointer (or, commonly, a return address) to subvert its execution, and direct it at the attacker-controlled input. An additional trick - Return Oriented Programming (ROP) is often used to direct execution to snippets of code which already exist in the program, but run in an attacker controlled manner. This method, which is somewhat like biological DNA splicing and recombination, defeats data execution prevention methods, such as ARM's XN bits. A lengthy discussion of past exploits and ROP methods can be found in the [Android Hacker's Handbook](#). It should be noted that not all exploits necessarily involve code injection - some are much more simple and elegant (for example, the "WeakSauce" exploit for HTC One phones, discussed in [the book's companion website](#)).

To paraphrase a quote attributed to Donald Rumsfeld - there are "known unknowns" - those are essentially the 0-days which were unknown, but have been discovered - and patched - but there are also "unknown unknowns". The latter are the 0-days which are likely to exist, but have not been discovered yet, or - worse - have been discovered, but not publicized yet. Any hacker uncovering a 0-day in effect obtains a skeleton key to all Android devices vulnerable to that particular issue. A malicious hacker can incorporate this into powerful malware, or not even bother, and directly sell it on the open market. Though not as lucrative as iOS exploits, Android 0-days can fetch anywhere between \$50,000 and \$500,000 dollars - depending on vector (local/remote) and impact.

Security Aspects of Rooting

Because a boot-based rooting method requires user intervention, and/or connecting the device to a host, it is generally not considered to be an insecurity of the Android system. It does, however, leave a clear attack vector for an adversary who gains possession of the device. This could be an issue if the device is lost, stolen, or just left outside one's reach for a sufficient amount of time. It would take a skilled attacker no more than 10-20 minutes to root a device, steal all the personal data from it, and leave a backdoor or two. This is why most bootloaders are often locked, and while an unlock of the bootloader is possible, it will force a factory reset and erasure of all personal data - Once the bootloader is unlocked, however, the device *is* vulnerable (unless the bootloader is locked again).

Exploitation attacks are even simpler in the sense that they do not require the user to manually divert the system boot process. In fact, these attacks require no user intervention at all. Therein lies their advantage (for those looking for a simple "1-click" root method), but also their great risk, as they can be carried out without the user's knowledge, often when installing a seemingly innocuous app, which like the proverbial Trojan horse compromises the entire system.

Exploitation attacks are even more dangerous when they are HTTP-borne. When the vulnerability exploited, or part thereof, involves the browser, it suffices to visit a malicious website - or inadvertently access some content from it (for example, through an ad network), for malicious payload to target the browser, and gain the initial foothold on the device. Indeed, sophisticated malware consists of multiple payloads injected over several stages, initially obtaining remote execution, then followed by obtaining remote root.

What follows is that rooting the device can, in fact, be dangerous, if not carried out through trusted sources: When an eager user downloads a rooting utility, whether one-click or tethered, if the download source is not a trusted one, it could be hard - virtually impossible - to detect additional payloads or backdoors which may be injected by such utilities. Less than proper tools may jump on the chance to also change system binaries or frameworks, for example disabling the Dalvik permission mechanism for malware purposes. Malware

could possibly inject a rootkit all the way down to the Linux kernel, though most would probably not put that much effort when it's fairly trivial to hack the higher layers. Somewhat ironically, some of the SuperUser applications themselves had vulnerabilities in the past, which enabled rogue applications to detect a rooted device, and escalate their own privileges through the applications (q.v. CVE-2013-6774).

The last, but hardly least impact of rooting a device one has to consider is that on applications - Android's Application content protections disintegrate on a rooted device: OBBs can be read by root, as can the keys to ASEC storage. Application encryption likewise fails, and though hardware backed credential storage offers some resistance, its client processes' memory can easily be read (via ptrace(2) methods and the like. DRM solutions also fail miserably. Unfortunately, there's no foolproof way of detecting a rooted device from a running application, and refusing to execute on one.

Arguably, the same can be said for Jailbroken iOS - after all, Apple's fairplay protections and application encryptions, though stronger than Android's, are equally frangible. Yet one has to keep in mind that iOS only has an exploitation vector (with an ever increasing level of difficulty in between releases), whereas most Android devices do allow Boot-to-Root. Coupled with the ease of Dalvik bytecode decompilation, this poses a serious concern for application developers.

Summary

This chapter attempted to provides an overview of Android's myriad security features, both those inherited from Linux, and those which are specific to Android and mostly implemented in the Dalvik level. Special attention has been given to the Android port of SELinux - which, though currently not in full effect, is already adopted by Samsung in KNOX, and is likely to play a larger part in upcoming releases of Android.

While trying to be as detailed as possible, this review is by no means comprehensive. The interested reader is referred to Android Security specific books, such as Nikolay Elenkov's [Android Security Internals](#).